CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

SPICE System Programmers Guide

Robert Sansom

21 Aug 84

Abstract

This document provides an overview of how to write programs which use the ACCENT operating system. Thus it explains how to use the server processes running under Spice, how to create your own processes and how to talk between your own processes. It does not attempt to explain how to make modifications to the operating system except for giving the procedure for constructing a new system image.

Spice Document S159

Keywords and index categories: Accent, Manual

Location of machine-readable file: [cmu-cs-spice]/usr/spicedoc/manual/spiceprogram/sysprog

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

Table of Contents

1 Introduction

In this document I aspire to provide sufficient information and explanation so that you, dear reader, can go ahead and use the features of the SPICE System and the Accent Operating System Kernel. I also hope that this document will reduce the learning time that I, for one, experienced when trying to understand how to use all the Accent facilities. If you find that I have omitted some vital point please feel free to tell me about it and I shall do my best to include it in a future version of this document.

I apologise in advance for the way in which this document is so orientated towards Pascal. This was historically due more to necessity than choice for the operating system and its interfaces were almost entirely written in PERQ Pascal. Now that SPICE LISP is available there should be a parallel version of this manual providing the LISP view of the world. The interfaces for C should be almost identical (allowing for language syntax) to the Pascal interfaces.

The following sequence of sections is meant to be fairly logical in that I will try and build up a coherent picture of the interprocess communication and process management facilities of Accent. To this end I have inserted a chapter describing the SPICE utility Matchmaker after the chapter on message handling. The later chapters mostly present the features of Accent in terms of a Matchmaker provided procedural interface.

1.1 Documentation

If you want a general overview of Accent you should read *Accent: A Communication Oriented Network Operating System Kernel* [Richard Rashid and George Robertson, *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981]. I summarise the main points of the operating system design in the next section.

The most important document to have at hand whilst reading this document is the *Accent Kernel Interface Manual* [Richard Rashid, 1982]. One of the aims of my document is to explain to the layman all the details contained in that manual. Another operating system document is the Perq Systems manual *Theory of Operations.*

Other useful documents are *Matchmaker: A Remote Procedure Call Generator* [SPICE Document S129], *User Manual for KRAUT - The Interim Spice Debugger* [SPICE Document S156], *User Interface to the Sapphire Window Manager, Application Interface to the Sapphire Window Manager* and *Sesame: The Spice File System* [SPICE Document S140]. These documents are all to be found in the SPICE *Programmers Manual.* In addition you will certainly need access to various Accent sources which will be found on the

/usr/spice account on the CFS VAX. The things that you are most likely to need will be found in *libpascal* and can be obtained by doing an update on the logical name *libpascalsrc*.

In Appendix A I have listed all the routine specifications that I use or describe in this document. These have been copied verbatim from the corresponding source files.

2 Operating System Overview

2.1 Basics

Accent is an operating system supporting multiple processes communicating via messages. There is no shared memory between processes and each process has its own virtual address space. Ports are the basic Accent object. They are used primarily as the addresses to which messages are sent. But they also play the role of naming system objects, such as processes, virtual memory segments and so on.

Ports are protected kernel objects. Within a process the name of a port is local to that process and only the Kernel can map local names for ports into kernel-wide names. The primary reason that this mapping is done is so that the kernel can protect rights on ports. A process can only gain rights (send, receive or ownership) on a port by appropriate mechanisms to be discussed later. A process cannot just make up a name, where a name is an integer, for a port and hope that it will be a valid name.

2.2 Processes

The Accent kernel performs only a small part of traditional operating system functions. It is left upto various server processes to provide the remaining functionality. The various servers are as follows:

- **System Init:** The process which is initially set running. It starts up the next four processes.
- **Time Server:** Obtains the current time over the ethernet and provides timing services for any other process.
- **Sesamoid:** The file server. This is an interim version of the full Sesame file server and provides file read and write functions as well as maintaining directories.
- **Environment Manager:** Manages SearchLists and facilitates the sharing of profile variables.
- **Startup:** The last process started up by the system init process. It brings up all the rest of the Server processes.
- **Sapphire:** Window manager for SPICE. Manages the PERQ's display and is capable of executing complex graphics commands.

- **TypeScript:** For processes which have traditional teletype style interfaces, typescript manages their input and output. It stores up some amount of i/o so that if you shrink and then enlarge a window you don't lose what was initially in the window.
- **Tracker:** Keeps tabs on the PERQ's Puck.
- **Procman:** The Process Manager. Does most of the handling of processes at a higher level than the kernel. Provides most of the hooks for debugging processes. Keeps a list of what processes are running.
- **Message-Name Server:** Called the network server or the MsgServer in some descriptions of the system. It provides transparent, remote inter-process communication over the ethernet. Additionally it functions as a primitive name server, both for local and remote name services.
- **Netserver:** Should be called the etherserver as it handles all of the dirty work involved in using the ethernet.
- **Shell:** The user's process. Functions as a simple command interpreter and allows the user to run and control other processes. You can have multiple shells running simultaneously. Each shell has its own window onto the screen.

3 Accent on the Perqs

3.1 Getting Started

Having found a working PERQ and switched it on you should now wait a couple of minutes to let it warm up. To boot the Accent Operating system you should press the boot button located on the back of the keyboard. Accent will announce itself. The system initialisation program then starts creating various Accent system processes and eventually the process manager is started. One Sapphire window will be created for the process manager. Another window, called the Icon Window, will be created to hold Icons representing the windows about which Sapphire knows. A final window will be created in which your initial shell (command processor) will run.

The PERQ disc is divided up into five or six partitions all named as */sys/name* and to begin with you will probably find yourself in partition */sys/spice.* Alternatively, if your machine has a name, the partitions will all be named as */your-machine-name/partition-name.* Using either of */sys/...* or */machine-name/...* will give the same result. The *spice* partition is where you will find all the Accent OS code and is not where you should create files as a user. The place for the latter is */sys/user.* Before moving to this partition by typing *path /sys/user,* you should add the directory */sys/spice/libpascal* to your search list by typing *setsearch libpascal.* Your search list will now contain */sys/spice* & */sys/spice/libpascal* and these should be sufficient to give you access to Accent utilities.

3.2 Writing Pascal Programs

If you are not wanting to use any Accent Operating System calls and just want to write an ordinary Pascal program then this is the last section of this document which you should read. Having typed in and edited your program using either the standard system editor or *Hemlock*, the Spice Lisp editor, you can compile it by typing *compile <name>*. By default the compiler produces files containing symbol table information. These are essential for debugging a program with Kraut. To get a RUN file you link the SEG file with *link*. To run the executable code type its name. Debugging means starting up Kraut. The debugger is implicitly invoked when an error occurs in your program. It can be invoked explicitly either by typing *debug <process number>* to another shell where the *process number* is the identity of the process executing your program or by following the name of the run file with a caret (eg. *<name>↑*).

3.3 Imports/Exports

PERQ Pascal provides you with a very simple modularization facility. Program modules can be compiled separately and brought together at link time. One module must be the *Program* module, all the rest are subsidiary. In a *Module* you can export declarations and routine specifications. These should be placed in the *exports* section, everything else in the module should be in the *Private* section. To import specifications exported by another module, you should write at the beginning of your *exports* or *private* section (whichever is appropriate) *imports <module name> from <file name>*. The typical convention is that *<module name>* and *<file name>* are the same and that the compiler appends *.pas* to the file name and looks it up in the current search list. The one exception to this convention are modules generated by Matchmaker where the file is called *<module name>USER* (eg. *TimeUser*). For more details you should see the PERQ Pascal manual and also the examples at the end of this document.

3.4 Importing Accent Modules

To start writing programs using features of the Accent Operating system, you must start importing modules which provide declarations of types or constants used in Accent and specifications of functions which are Accent system calls or calls to other server processes. The most important modules are those that provide the interface to the kernel. These are *AccentType* which gives lots of definitions for system objects (see later), *AccCall* which gives the fundamental kernel calls and *AccInt from AccentUser* which gives the kernel calls to which there is a message passing interface. The module *PascalInit* provides those global variables which are accessible to the user and are needed for communicating with server processes. You will almost certainly need to import it. Also worth looking at is *SaltError* which provides translation of Accent system error codes.

Library modules are available either as *stubs* or as complete modules. A Stub contains only the part of the module which is exported by that module. Typically this means that the stub is a lot smaller that the whole module thus saving quite a lot of disc space. Compilation should also be a bit quicker since the files that are read in by the compiler are that much smaller. You can write your own modules in the stubs plus body style by having the body included into the *private* section of the modules with the appropriate compiler directive (*$include*).

3.5 The Linker

The Accent Linker can be quite a powerful beast. This section is meant to give some guidance on how to handle it. The normal case, which is just linking some seg files into a run file, was mentioned above. To make loading of run files more efficient one can specify the switch *include* which means that the object code from the files is actually included in the run file instead of there being pointers from the run file to the seg files. Of course one has to pay the price of extremely big run files! Similarly the switch *syminclude* means that the SYM and QMAP files produced by running the compiler with the *scrounge* switch are actually copied into the run file. You do not need to use the *syminclude* switch as *KRAUT*, the debugger, does a better job of locating SYM and QMAP files.

The most useful *advanced* feature of the linker is its ability to copy SEG files out of RUN files. This enables you to rebuild a RUN file from an old RUN file and, for instance, just one new SEG file. All the other SEG files necessary for building the new RUN file are obtained from the old RUN file. An example will be the best explanation. I want to make changes to the *startup* program which is the first program, apart from the boot core image, that is run by Accent on start up. I just want to make changes to the *procman* module which is part of the startup program. Also the original *startup.run* was linked on someone else's PERQ and I want to link it with my *libpascal* files and not his. In this complicated case the linker command should be:

```
link -library=/sys/spice/libpascalinit -verbose -forceload -include
        procman, /sys/spice/startup.run -main -include
        ~/sys/spice/startup.run
```

- The *library=/sys/spice/libpascalinit* means that libpascal SEG files will be taken from the local *libpascalinit.run.*
- The *verbose* switch gives me more information about what is going on so that I can check that the link is correct.
- *forceload* ensures that all files specified on the command line are actually loaded.
- *include* ensure that the body of *procman.seg* is copied into the output RUN file.
- */sys/spice/startup.run -main -include* says that *startup.run* contains the main module and the remaining SEG files all of which should be included in the resulting run file.

- Lastly the twiddle ~ says that the output run file should be called */sys/spice/startup.run.* As far as I can tell it is safe to overwrite the old RUN file.

4 Message Handling

4.1 Accent Primitives

To send or receive Accent IPC messages just call the kernel routine *send* or *receive* and pass it a message header (see below) as a var parameter. On calling *send*, the message which is associated with the header will have all its data sent. On calling *receive*, the message received will be placed in the message associated with the message header. Options and other parameters to send and receive are described in the Kernel Interface Manual and to start with you should just use the obvious defaults. These are 0 for *MaxWait* (implying wait for ever), *WAIT* for send and *RECEIVEIT* for receive. (The *PortOpt* parameter of receive is described below.)

Before a message can be sent, you must construct a message in memory. The message can be represented in a contiguous block of memory or by a block containing pointers to other blocks. Only one level of indirection is allowed.

To receive a message, a block of memory must be allocated for the header and contiguous data part of the incoming message. If the incoming message has pointers to other blocks, the kernel will find unused portions of your address space and place these out of line blocks in such portions. There is no way to specify where these message pieces go.

In general, the kernel does not change the message representation: if a message is sent with pointers, it will not be reformatted to be contiguous at the receiver side. Furthermore, page alignments are normally maintained except for the header and contiguous part of the message, whose location is specified in the *receive* call.

4.2 Message Construction

The difficult and tedious part of message management is the constructing and filling in of all the fields that a message must contain. There are two distinct parts to a message, the header and the data. As seen in the diagram below the data consists of an alternating sequence of data item description and actual data.

```
-------------------------
|        Message        |
|        Header         |
|***********************|
|   Item 1 Description  |
|-----------------------|
|   Item 1 Data         |
|=======================|
|   Item 2 Description  |
|-----------------------|
|   Item 2 Data         |
|=======================|
|         etc.          |
|                       |
-------------------------
```

4.2.1 Message Header

In the module *AccentType* you will find the predefined types *Msg* and *ptrMsg*. The latter is simply a pointer to the former. They provide the structure for an Accent message header. The most important fields in the header are *LocalPort* and *RemotePort*.

On sending a message the *RemotePort* must contain the port to which you wish to send this message. (See later sections for more discussion on Ports.) The *LocalPort* field contains a port which you wish the recipient to use as a reply port (it may be set to *NullPort*, which is defined in AccentType to be 0). The recipient can obtain this port from the *RemotePort* field after his call to receive returns successfully. What happens is that the local port on the sender's side gets mapped to the remote port on the receiver's side and vice versa. Thus the receiver will also be able to find out upon on which port the message was received.

On receive the *LocalPort* specifies on which of the ports, which your process owns, the process should wait for a message. This behaviour only occurs if you set the third parameter (*PortOpt*) to *LOCALPT.* If you set *PortOpt* to be *ALLPTS* then the *LocalPort* field of the header can be left undefined and all ports to which you have access will be examined for messages. The *Localport* field will then be set by the kernel to tell you which port the message came from.

Other relevant fields of the message header are:

• *SimpleMsg*	See later
• *MsgSize*	The size of your message (in bytes)
• *MsgType*	Whether this message is *NORMALMSG* or *EMERGENCY*
• *ID*	For your convenience - use by convention

Note that the *MsgSize* field must be filled in on both *send* and *receive.* If you *receive* a message larger than *MsgSize* or larger than the amount of space that you have allocated to receive the message data, a *NotEnoughRoom* error will occur. Similarly the *MsgSize* on *send* should accurately reflect the total size. You should not leave any unused space after the last data item in the message. In both cases the conventional technique is to declare a record representing your message and then use the Perq Pascal intrinsic *WordSize* to fill in the *MsgSize.* Don't forget to multiply by two because whereas *wordsize* returns the number of 16-bit words, the header field requires you to provide the number of bytes.

4.2.2 Data

Almost any number of data items can be included in an Accent message (the limit is about 1K of inline data). You will find it most convenient to construct a record type for yourself representing the message format that you wish to use. (See the examples at the end of this section and in Appendix B.) You can then use the *wordsize* intrinsic of PERQ Pascal to fill in the message size field of the header as above.

The major distinction between different data items is whether they are inline or not. Inline data is suitable for small amounts of data, the size of which you know in advance. Out-of-line data is good for large amounts of variable sized data. Out-of-line data is referred to by a pointer contained as inline data within the message. The Accent kernel ensures that all data pointed to is logically sent across to the other process. If you use ports (more later) or pointers in the inline data then you must set the *SimpleMsg* field in the header to be *FALSE* otherwise it should be set to *TRUE.*

For each data item you have to describe various characteristics. You can do this either within the confines of one 32-bit word or by expanding the attributes in several words. In either case the first 32bit word should be of type *TypeType* (yes - that really is its name). (See Appendix A for the PASCAL declaration of *typetype.*) The boolean field *LongForm* of your *typetype* value says whether you are using the short form or the long form. The long form means that you need to use two extra 16-bit words followed by one extra 32-bit word in your type declaration for message items.

In either case the most important field is the *TypeName* field. This provides a uniform coding for the type of the data that you are passing. Thus the remote process will be able to check that it has received the types of data it was expecting. Additionally the kernel will trap any data of a Port type (again see below for more details) to convert references to ports from being local to this process to being local to the other process. Note that for the long form the *TypeName* field is the first extra 16-bit word.

You are also responsible for filling in the size of each element (*TypeSizeInBits*) and the number of elements you are passing (*NumObjects*) for each data item. Typically the latter figure will be one. However if you are passing for instance a string then you have two options (taking the default string length of eighty as example). You can either pass 82 elements of *TypeChar* each of length 8 or one element of *TypeString* which is of length 652 (8*82). (A string of nominal length 80 takes up 82 bytes because there is one leading byte giving the string's actual length and one trailing pad byte to bring it upto a 16bit word boundary.) Note that for the long form the *TypeSizeInBits* field is the second extra 16-bit word and *NumObjects* (called *NumElts* in the *Kernel Interface Manual*) field is the extra 32-bit word.

The last attribute of a data item is *Deallocate.* This is only relevant to out-of-line data. If it is set, then when the message has been successfully sent, the local copy of the data sent will be disposed. The other field that has been implicit in all the above discussion is, of course, the boolean *Inline* whose use should by now be obvious.

4.3 An example

I assume that the modules *AccentType* and *AccCall* are imported. First I define the message format:

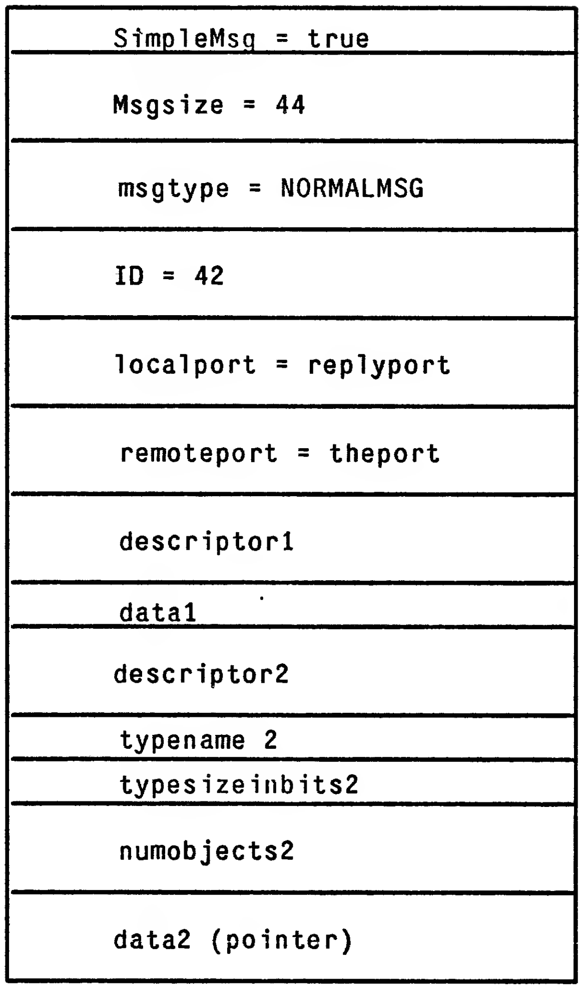
```
var mymsg : record
                head             : Msg;
                descriptor1      : typetype;      { This is a short form }
                data1            : integer;
                descriptor2      : typetype;      { This is a long form }
                typename2        : integer;       { and these are }
                typesizeinbits2  : integer;       { the three }
                numobjects2      : long;          { extra fields. }
                data2            : ↑array [0..99] of integer
            end;
```

Next fill in all the fields of the message:

```
with mymsg do
        begin
        head.simplemsg   := false;        { because we have out ofline data }
        head.msgsize     := wordsize (mymsg) * 2;
        head.msgtype     := NORMALMSG;
        head.ID          := 42;           { my choice }
        head.localport   := replyport;  { A port for the reply }
        head.remoteport  := theport;      { The port to send to }

        descriptor1.deallocate   := false;
        descriptor1.longform     := false;
        descriptor1.inline       := true;
        descriptor1.typename     := TYPEINT16;
        descriptor1.typesizeinbits := 16;
        descriptor1.numobjects   := 1;

        data1            := 678;

        descriptor2.deallocate   := true;
        descriptor2.longform     := true;
        descriptor2.inline       := false;
```

```
typename2         := TYPEINT16;
typesizeinbits2   := 16;
numobjects2       := 100;

new (data2)
end;
```

The message actually constructed looks something like:



Finally we can actually send the message:

```
return_value    := send (mymsg.head, 0, WAIT);
```

The receiving process can just do:

```
return_value    := receive (amsg.head, 0, ALLPTS, RECEIVEIT);
```

5 Matchmaker

5.1 Overview

By now you ought to have come to the conclusion that all the work required to create a suitable message for sending is extremely tedious and prone to errors. You are of course quite correct. This is where Matchmaker comes into the picture. Matchmaker will do most of the mechanical work for you and will present a Procedure Call-like interface for each type of message that you wish to send. Matchmaker can generate PERQ Pascal, C and SPICE LISP interfaces.

Matchmaker generates code to provide *Remote Procedure Calls.* Remote Procedure Calls look like ordinary procedure calls to the user but are implemented by the underlying message passing mechanism. They are called *Remote* because the call is from one process to another, unlike normal procedure calls which are local to one process.

From one definition file, written in its own source language, Matchmaker will generate two interface modules. One corresponds to the user or client side of the business and the other to the server side. Thus the user will call a procedure in the *user* module which he hopes will do something useful for him. This procedure constructs a message, fills it in appropriately and sends it off to the server. The server must receive the message and then call the managing procedure in the *server* interface module. This takes the message received, dissects it and calls the appropriate procedure in the server. When the server has done the work required of it, it returns to the *server* interface module which constructs a reply message. This message is then sent back in turn to the *user* module which receives it, extracts the returned values and finally returns to the user. Phew!

For further explanation see the *Matchmaker* manual. For a picture see the end of this section.

5.2 Definitions Files

The Matchmaker definitions files that you will need to write should be fairly simple. The first line of the file should be the name and numerical identity of your subsystem. Thus:

```
        Interface Example = 666
```

(There exist conventions for numbering subsystems but you only need worry about them if you are writing a system server.) The next few lines should give stylised details of all the types that you intend to pass across the interface. For more details see the next section or the Matchmaker manual. After a few irrelevant fields we come to the meat of the specification - the individual procedural interfaces. Typically you should specify them to be of type *Remote_Procedure* in order to generate functions which send request parameters to a server and receive reply parameters. The declaration looks like:

```
Remote_Procedure CallName (ArgList) : ValueType;
```

The *ValueType* should probably be *GR_Value* which means that the function returns the Accent result type *GeneralReturn.* For each call one parameter should identify the port to which this message is to be sent. This parameter is specified by the keywork *RemotePort.* Further parameters are values to be sent in the message (the default), returned if *out* is specified or sent in both directions if *inout* is specified. For example:

```
Remote_Procedure Foo
                ( RemotePort  ServerPort : port;
                              Data       : integer;
                  Out         Result     : long)   : GR_Value;
```

5.3 IPC Type Information

The main input to Matchmaker apart from the syntax of the remote procedure calls is the IPC typing information for the values passed across in these calls. Matchmaker has predefined common types - for instance a *Short* is converted to the IPC type *TYPEINT16* with a type size of 16 bits. For other more esoteric types you should declare your own IPC type information. This can be done at the beginning of the definitions' file, for instance:

```
Type string   =  array [82] of (TypeChar, 8);
```

This means that wherever the type *string* occurs in a routine specification then Matchmaker will fill in the IPC information such that the object has IPC type *TYPECHAR*, type size in bits of 8 and that there are 82 elements. (Note that a string of *N* characters takes up *[(N + 2) DIV 2] * 2* bytes - one extra byte for the string's size plus a possibly extra pad byte.)

5.4 The User Side

About all you need to do is to import the user module and then call the functions of the server as if they were local. But before you use the interface you must initialise it by calling the procedure *Init<name>* with a port as the parameter. This port will be used as the reply port for requests to the server implementing the functions. Passing *NULLPORT* to the *Init* procedure means that a new port will be allocated within the user module. For an example see the program in Appendix B.

5.5 The Server Side

Writing the Server side of a Matchmaker interface is not quite as easy as using the User side. On the Server side you have to do more of the work. Your main routine should first receive a message that you want to be interpreted by the Matchmaker interface. Then it should call the function *<system>server* exported by the Matchmaker Server interface. This routine will parse the received message, check the types of the arguments and then calls the routine, that you have provided, which will do the real work. It is probably best if you put the real routines in a separate module which can then be imported by the Server module. On returning from the real routine the Server routine will package up a reply message before control is returned to the top level. The main routine should then send this reply message for which the Matchmaker User side will be waiting. The Server function will return a boolean value which says whether the received message was understood by the Server interface.

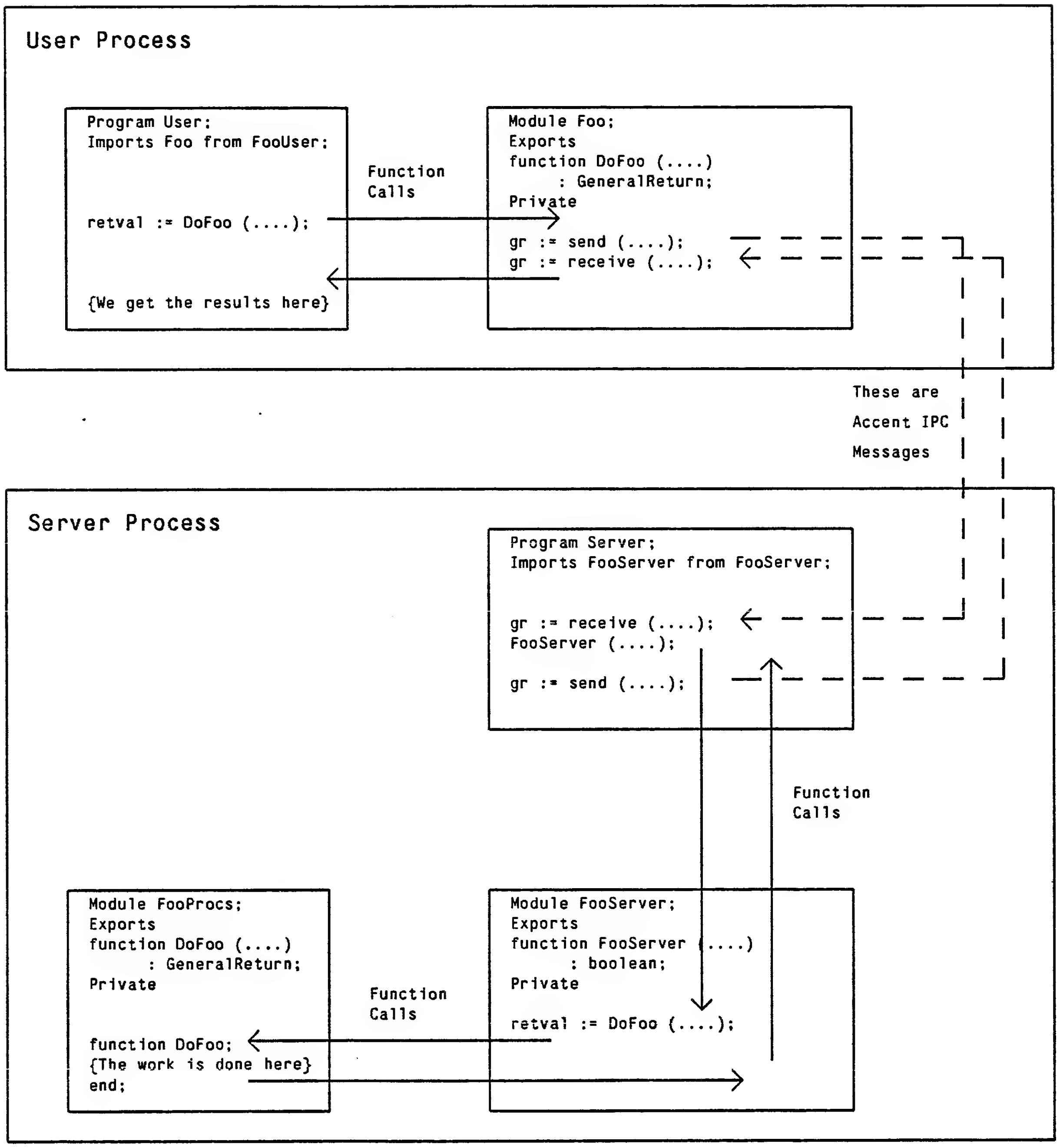
The Server routine is called with two parameters, one being a pointer to the message just received and the other a pointer to the reply message. Thus is is probably easier to declare two variables for the respective pointers. You must ensure that they point to data areas which are large enough to receive the largest expected message and to contain the biggest possible reply message respectively. In addition the data area used for the incoming message should consist of an Accent Message header followed by the actual data area and you should fill in the *MsgSize* field of the header to reflect the size of message you are willing to receive. Similarly if you are likely to receive messages from more than one source then you can specify the *LocalPort* field of the incoming message to be, say, your special server port.

Here is an example of what the main routine could look like:

```
Program example;

imports AccentType       from AccentType;
imports AccCall          from AccCall;
imports ExampleServer    from ExampleServer;    { exports function ExampleServer }
imports AccInt           from AccentUser;
imports PascalInit       from PascalInit;       { For the Name Server Port }
imports MsgN             from MsgNUser;         { For Name Services }

Type
  InMsgPtr    = ↑InMessage;
  InMessage   = record
                    Head : Msg;
                    Data : array [100] of integer;
                end;
  RepMsgPtr   = ↑RepMessage;
  RepMessage = array [14] of integer;

Var
  ServerPort      : port;
  RequestMessage : InMsgPtr;
  ReplyMessage     : RepMsgPtr;
  ok               : boolean;
  GR               : GeneralReturn;

Begin
  new (RequestMessage);
  new (ReplyMessage);

  { Allocate the serverport and check it in. }
  GR := AllocatePort (KernelPort, ServerPort, DefaultBacklog);
  InitMsgN (NullPort);                { Initialises the Name Server Interface }
  GR := CheckIn (NameServerPort, 'ServerName', NullPort, ServerPort);

  while true
  do begin
     RequestMessage↑.Head.MsgSize    := WordSize (InMessage) * 2;
     RequestMessage↑.Head.LocalPort := ServerPort;
     GR := receive (RequestMessage↑.Head, 0, LOCALPT, RECEIVEIT);
     if GR = success
     then begin              { Process the Request. }
          ok := ExampleServer (RequestMessage, ReplyMessage);
          if ok              { Request was correctly processed. }
          then GR := send (ReplyMessage↑.Head, 0, WAIT)
          end
     end
End.
```

5.6 What it all looks like

This picture shows the modules, processes, functions calls and messages involved in executing the remote procedure called *DoFoo.* The Matchmaker generated modules are *FooServer* and *Foo.* The latter module is contained in the file *FooUser.*



Example of the mechanics of a Matchmaker interface

6 Ports and the Message-Name Server

6.1 Ports

A port is a protected kernel object. All ports are local to a particular process and are mapped by the kernel into processor-wide ports. The most important ports that you ought to know about are the *KernelPort* and the *DataPort*. You obtain access to the definitions of these ports by importing the file *AccentType*. The *KernelPort* represents your process' right to make service requests of the kernel. The *DataPort* is used by the kernel to send messages to you. As a user process you also have access to various system ports, in particular the *NameserverPort*, the *TimePort* and the *PMPort*, the definitions of which are to be found in the module *PascalInit*. Additionally you can access the ports *SapphPort*, *TypeScriptPort*, *UserWindow* and *UserTypeScript*, which allow you to create or manipulate windows and typescripts.

To create a new port you should use the call *AllocatePort*, the first parameter of which must be your *KernelPort*. The second parameter is the name for the new port and the last parameter is the backlog, just set this to zero which will give you the default backlog.

As previously mentioned, ports are used for sending and receiving messages. Thus with each port the kernel associates send and receive (and ownership) rights. At any one time only one process can have receive (or ownership) rights but multiple processes can have send rights. When you create a port your process will possess all three rights. The ownership right is normally coupled with the receive right. However it's purpose is to provide some amount of failure protection. If ownership and receive rights are associated with two different processes, then one process will be informed by the kernel if the other one dies. This information is in the form of a port death emergency message (see section 10 for more information on emergency messages). The emergency message contains the right which was associated with the freshly dead process.

There are two ways of letting other processes have access to your ports. These are discussed below.

6.2 Sending Ports in Messages

If you have *a priori* knowledge of a port belonging to another process (for example by importing some system module), then you can send a message directly to this port (assuming you have send rights on it which is normally the case). In this message you can include a port (see previous sections on message handling and typing of message items). The default rights associated with a port sent in a message are send rights. To send a port with receive or ownership rights you must specify the type of message item to be *TYPEPTRECEIVE* or *TYPEPTOWNERSHIP* respectively.

Once another process receives send rights on a port on which you have retained receive rights, it will be able to send messages to you.

6.3 Message-Name Server, the Name half

A more common case in the SPICE client-server environment is that you don't know anything about another process with which you wish to communicate other than some textname identifying it. This is where the Name half of the Message-Name Server (hereafter the Name Server) comes into the picture. To find a port on which you can communicate with another process you must call the function *LookUp.*

All Name Server routines can be obtained by importing *MsgN from MsgNUser.* To communicate with the Name Server you must have access to the *NameServer* public port which is obtained by importing the file *PascalInit.* The value of the *PortsId* parameter returned will be the identity of the port upon which the other process is willing to receive messages. You can then initiate two-way conversation as described above.

You will be (or should be) wondering by now just how this port that you have just obtained was registered with the Name Server in the first place. The registration is done by a *CheckIn* call. *CheckIn* allows you to associate a string name with a port for which you want to give other processes receive rights.

Name Server *CheckIn* and *CheckOut* routines require an extra port parameter which is meant to provide security. On *CheckIn* you quote this extra port as authorizing operations on the string name and port you are checking in. This port can only be deleted from (or changed in) the Name Server's tables if the *CheckOut* (or new *CheckIn*) call also contains the authorization port. Typically, at present, since we aren't (though perhaps we ought to be) worrying about security, you should just set this authorization port to *NULLPORT.*

6.4 Message-Name Server, the Message half

One excellent feature of Accent is that remote communication over the network connected to your PERQ is virtually transparent. Obviously there are quantitative differences (eg. longer delays), and there is a completely different failure model, but I shall not worry about these details here. The main duty of the Message half of the Message-Name server (hereafter the Message Server), apart from sending and receiving messages on the Ethernet, is to map ports local to the processor to ports which have a unique identity over the network. Thus when you communicate with a remote process you will be going via both your Message server and the other processor's Message server.

In practice, you should not worry about the presence of the Message server and should continue to use the message passing and port management features of Accent as though all processes were local. The current Name Server which, as you have seen, is incorporated into the Message-Name server functions as a primitive network Name Server, thus you don't even have to worry whether the string names you look up map into local or remote ports.

6.5 Accent style IPC under UNIX

If you want to have UNIX processes communicating with Accent processes then you should know that there is a UNIX subroutine library which will allow you to use Accent style IPC under UNIX. The subroutines map Accent messages onto the corresponding CMU UNIX IPC messages. The CMU UNIX IPC is the precursor of Accent IPC and thus the two are quite similar in many respects. However it is easier if you use the subroutines which hide the differences. More documentation is available as *AccUnix - Accent style IPC under UNIX* by yours truly (SPICE document S160).

Futhermore there is an Accent style Message-Name server available under UNIX. This means that you can just as easily do remote IPC between an UNIX process and an Accent process as between two Accent processes. Most of the Computer Science Department Vaxes have one of these Accent style Message-Name servers running on them.

7 Memory Management

Unless the kernel is told otherwise, an arbitrary virtual memory location is considered invalid and may not be referenced. A normal user process has access to three primitives for rectifying this situation. *ValidateMemory* marks a given or a new part of a process' address space as valid. References to data in the newly validated memory will succeed and return zeroes. The parameter *CreateMask* specifies the address alignment of new memory; a value of -1 implies no particular alignment. *InvalidateMemory* performs the inverse of *ValidateMemory*. *MoveWords* moves data from one place in a process' address space to another. For more details on the above three functions see the *Kernel Interface Manual.*

The most important memory management objects are segments, which can be created, read, written or destroyed. There are four types of segments. A *permanent* segment is allocated on disc and survives machine crashes. Thus if you want to store data, but don't want to use the full Sesame file system, then you can manipulate permanent segments to achieve your own ends. *SegPhysical* segments are contiguous regions of real memory and are used to handle devices. *Temporary* segments are similar to permanent segments except that they disappear when all the processes having access to them die. *Imaginary* segments are segments ids without storage. References to data within them are forwarded to an *ImagSegPort* which has previously been associated with the segment id. As you have no doubt guessed by now, use of imaginary segments is fairly esoteric!

To access *permanent* segments, you need to have access to the ports associated with the disc partitions.

For *segphysical* segments, you need to have access to a specially protected port. These ports are not available to normal user processes. For the other types of segments your kernel port suffices to give access.

All the calls to manipulate segments do so in terms of whole pages. Thus even if you only want to write two bytes to a segment you will end up effectively writing the whole page of the segment. Of course you won't write the rest of the page with new data. When you create segments you have to specify the initial and maximum size of the segment in terms of pages. Note that these sizes are ignored for permanent segments since the kernel manages the disc space with a modicum of intelligence. For instance one can create a segment on disc and then write one byte to page 835. The segment will only take up two pages on disc, one for the header and one for page 835.

8 Sesame

8.1 Introduction

The Sesame system will eventually encompass not only a file service but also name, authorisation and archival services. You can find full details of the procedural interface to Sesame in the *Sesame: The Spice File System* chapter of the SPICE Manual. Currently in operation is a preliminary version (*Sesamoid*) of the file server. From now on when I write *Sesame* I probably should write *Sesamoid.*

In this section of the *Programmers' Guide* I briefly describe how to do simple management of files. The routines which I actually describe are from the module *PathName* which provides a non-primitive interface to the Sesame File Server. The first thing that you ought to know is that Sesame manipulates two types of names. *Absolute Path Names* always begin with a forward slash (/) and unambiguously name a file (assuming that the file exists). *Relative Path Names* are interpreted by Sesame relative to your current file system defaults. These defaults are stored by the environment manager but that need not worry you. Defaults include search lists and logical names - the use of which you should be familiar with as a SPICE user. Anyhow, given a relative path name in the appropriate context, Sesame will convert it to the corresponding absolute name.

8.2 PathName

From *PathName* the two important routines are *ReadFile* and *WriteFile. ReadFile* given a relative path name will read a file. The file data is read into memory. A pointer to the data and the number of bytes read are returned. In addition the path name you gave to *readfile* will be filled in with the absolute name of the actual file read. Thus this first parameter must be a variable. Don't forget that the *ByteCount* actual parameter must be a long variable.

WriteFile will both create a new file and overwrite a current file. The data that you wish to write to the file should be pointed to by the *FileData* parameter and the number of bytes to be written should be specified by the *ByteCount* parameter. Like *readfile*, *writefile* will overwrite the name passed in with the absolute name of the file actually written. In the long term Sesame will provide version numbers for all files. A write will always create a new version of the file leaving older versions untouched. Currently all files are of version 1; thus the name returned is something like /..../..#1. You must not try and specify the version number to *writefile* since you cannot write to a particular version, only to a *new* file. However for *readfile* the absolute name should include the version (ie. the #1). More details about these routines can be found in the *PathName* section of the *Pascal Library* of the SPICE Manual.

9 Process Management

9.1 Accent Primitives

If you are a glutton for punishment then you need only use the primitive functions provided by Accent and described in the Kernel Interface Manual. Using the *fork* primitive to create a new process means that you have to do all the hard work of initializing the new process' environment, especially the interface to Sapphire (the Accent display management facilities).

The other primitive functions besides *fork* are easier to use as they allow manipulation of processes already created. Most important is the function *terminate* which allows you to perform infanticide. You can also manipulate child process' priority, computation time limits and state (running or suspended). The debugger uses these facilities to manipulate processes being debugged.

As mentioned in the previous section there are two special ports owned by every process, the *Kernelport* and the *DataPort*. A process has send rights on its *KernelPort* and receive rights on its *DataPort*. The kernel has the corresponding receive and send rights respectively. When you fork a child it will be created with its own *KernelPort* and *DataPort*. The difference is that the parent process also has rights on these two child ports. Thus the parameters for *fork* must include two port variables. The kernel will create the ports and initialize these two variables. As a parent process you have send rights on both the child's *KernelPort* and its *DataPort*. To manipulate a process you must always quote its *KernelPort* - this represents your right to manipulate it. Thus since you know your own *KernelPort* as well as your children's you can manipulate yourself as well as them. In particular to fork a process the first parameter to *fork* must be your own *KernelPort*. It is possible to communicate to the child using its *DataPort* but this is not recommended. Instead, if you wish to send message to the child, the first thing you should have it do is to send you a message containing a port which it has allocated and on which it will receive messages from you.

When you fork a process you can pass it ports to which you have access to or which you have created in an array of ports. Use the type *ptrPortArray* to simulate a dynamic array of ports. The *PortsCount* parameter of *fork* should be set to the number of ports you have put in the array. Ports should be put in the array sequentially starting from zero. You may have wondered how your process initially gets ports to the nameserver, fileserver, etc. The ports are passed from parent to child when the child is forked. By convention, some of the ports are copied from the port array provided by fork into a set of port variables exported by *PascalInit*.

After a call to *fork* you can find out whether you are the parent or the child. The return value from the call will be *IsParent* or *IsChild* accordingly.

9.2 Process Manager

Higher level routines for process management are provided by the process manager (or ProcMan). This has a Matchmaker generated user interface hence you should import *PM from PMUser* to use it. Some Procman routines map almost directly onto Accent process manipulation primitives. Others do a lot more for you.

More details about the Process Manager can be found in the SPICE Server manual.

9.3 Spawn

The most useful Accent utility for process management is *Spawn*. Spawn exports three routines which enable you to do most of the process creation that you will ever need and in addition initialise the new process' environment for you. All three functions take, as their first two parameters, variables which will be the parent's access to the child's kernel and data port as described above.

The function *Exec* is the simplest way to run a program. The new process will execute the run file passed as the *ProcessName* parameter. The new process will inherit the caller's window and typescript and a copy of its file system connections.

The function *Split* effectively does a *fork* with the child sharing the caller's window and typescript and getting a copy of the caller's file system connections.

The function *Spawn* provides the full generality of process creation. I find it most useful, even though it is not well documented. In fact the shell itself uses *Spawn* to create a new copy of itself when you type **shell** to obtain a new shell. The new shell inherits the environment of the shell from which it was forked and it is *spawn* that manages this inheritance.

Now follows a description of the major parameters to *Spawn*. The *ProgName* parameter gives the name of the run file to be executed. The *ProcName* (don't get confused with *ProGname* - it's like the difference between stalaCtites and stalaGmites) gives the name that you wish to be associated with the new process. This name will appear in the process manager window. *HisCommand* allows you to pass over a command block to your child. The child will access this command block through the global variable *UserCommand* which is declared in *PascalInit*. The *SapphConn* parameter says whether you want the child to inherit the parent's window or not. There are in fact three options available for the *ConnectionInheritance*. If it is *Given* the new sapphire window and typescript will be taken from the following two parameters to *Spawn*, *pWindow* and *pTypeScript* respectively. If it is *NewOne* the the process manager will be asked to allocate a new window. The third option is *GivenReg* which is to our intents the same as *Given*. Similar options apply to the parameter *EMConn*, but you may as well ask for *NewOne* for all I know. The penultimate pair of parameters allow the parent to give to its child process send rights on ports. The ports being passed across should be placed in the array pointed to by *PassedPorts*, indexed from zero, and the number of ports should be stated in *NPorts*. The child gets access to these ports from the *ptrportarray InPorts*, the definition of which is contained in *PascalInit*. The indices of the ports in this array are the same as the indices that the parent provided in *PassedPorts*. The final parameter *LoaderDebug* turns on debugging of spawn itself and either the pascal or C loader.

One word of warning. It is probably better for the parent not to use the child's *DATAPORT* for sending messages to the child even though the parent has send rights on this port. If you try and send a message to the child's *DATAPORT* before the child had been completely initialised, then things are quite likely to go wrong and result in a fuzzy exception. You should either wait until the child sends the parent a message saying that it is running or just wait for some amount of time (I can't say how long). The optimal solution is to have the child send to the parent a message which includes a port allocated by the child. The parent can then use this port to send messages to the child and the child will know that the messages are coming from the parent.

10 Emergency Messages

10.1 Occurrence

Emergency Messages are high priority messages. Anyone can send them but my purpose here is to make you aware that you will quite likely receive them from the Accent kernel. The most important Kernel Emergency message is one notifying you of the deletion of a port upon which you had send rights (or one and only one of receive or ownership rights). This message will be sent when the process owning receive or ownership rights (or both) on that port dies. This will often occur when one of your child processes dies. Its death will cause the deletion of its *DATAPORT* and since you were given send rights on this port when you forked the child, you will be notified of this deletion. In fact this is the Kernel's only method of informing you that a process which you forked has died. There is no other mechanism and it is only the child's *DATAPORT* and *KERNELPORT* which name that child process.

10.2 Format

A Kernel Emergency message contains in its *ID* field a code representing of what you are being notified. The values of these codes are defined in the *AccentType* module. In addition the message data consists of a single port field (plus the associated *typetype* information) which will contain the port to which the message refers. The PASCAL declaration of an emergency message is something like:

```
type EmergencyMsg = record
                      head      : Msg;
                      porttype  : typetype;
                      deadport  : port
                    end;
```

11 The End of the Beginning

Well, it's your turn now. I suggest that you try writing some simple programs which use features of the Accent Operating System which interest you. If you run any difficulties please don't hesitate to ask a member of the SPICE group for help. Also you ought to have access to copies of both the *Users' Manual* and the *Programmers' Manual.* These overweight documents should answer a lot of the questions you have - even though it may take a bit of effort to find the solutions.

Good luck. May all who sail in her have a safe passage.

Appendix A. Type and Routine Specifications

A.1 From AccentType

```
type
        TypeType            = packed record
                            case integer of
                            1: (       TypeName           : Bit8;
                                       TypeSizeInBits     : Bit8;
                                       NumObjects         : Bit12;
                                       InLine             : boolean;
                                       LongForm           : boolean;
                                       Deallocate         : boolean );
                            2: (       LongInteger        : long )
                             end;

        Port                = long;

        ptrMsg              = ↑Msg;
        Msg                 = record
                                       SimpleMsg          : boolean;
                                       MsgSize            : long;
                                       MsgType            : long;
                                       LocalPort          : Port;
                                       RemotePort         : Port;
                                       ID                 : long
                             end;

        ptrPortArray        = ↑PortArray;
        PortArray           = array [0..MAXPORTS-1] of Port;

GeneralReturn               = integer;
const
        NULLPORT            = 0;
        KERNELPORT          = 1;
        DATAPORT            = 2;

const
        WAIT                = 0;
        DONTWAIT            = 1;
        REPLY               = 2;
type
        SendOption          = WAIT..REPLY;

const
```

```
        PREVIEW             = 0;
        RECEIVEIT           = 1;
        RECEIVEWAIT         = 2;
type
        ReceiveOption       = PREVIEW..RECEIVEWAIT;

const
        DEFAULTPTS          = 0;
        ALLPTS              = 1;
        LOCALPT             = 2;
type
        PortOption          = DEFAULTPTS..LOCALPT;

type
        SegID               = long;
        SpiceSegKing        = (temporary, permanent, bad, segphysical, imaginary);
        VirtualAddress      = long;
```

A.2 From AccCall

```
Function Send              ( var  xxmsg          : Msg;
                                  MaxWait        : long;
                                  Option         : SendOption
                           ) : GeneralReturn;

Function Receive           ( var                 xxmsg        : Msg;
                                  MaxWait        : long;
                                  PortOpt        : PortOPtion;
                                  Option         : ReceiveOption
                           ) : GeneralReturn;

Function MoveWords         (      SrcAddr        : VirtualAddress;
                             var  DstAddr        : VirtualAddress;
                                  NumWords       : long;
                                  Delete, Create : boolean;
                                  Mask           : long;
                                  DontShare      : boolean
                           ) : GeneralReturn
```

A.3 From AccInt from AccentUser

```
Function AllocatePort      (      ServPort       : Port      { Your KernelPort }
                           var    LocalPort      : Port;     { The port to be created }
                                  BackLog        : Integer
```

```
                        ) : GeneralReturn;

Function Fork           (     ServPort       : Port;      { Your KernelPort }
                        var   HisKernelPort  : Port;      { Child's KernelPort }
                        var   HisDataPort    : Port;      { Child's DataPort }
                        var   Ports          : ptrPortArray;
                        var   Ports_Cnt      : long
                        ) : GeneralReturn;

Function Terminate      (     ServPort       : Port;
                              Reason         : long
                        ) : GeneralReturn;

Function ValidateMemory (     ServPort       : Port;
                        var   Address        : VirtualAddress;
                              NumBytes       : long;
                              CreateMask     : long
                        ) : GeneralReturn;

Function InValidateMemory (   ServPort       : Port;
                              Address        : VirtualAddress;
                              NumBytes       : long
                        ) : GeneralReturn;
```

A.4 From SaltError

```
Function GRErrorMsg     (     GR             : GeneralReturn
                                             { The return code to be translated }
                        var   Msg            : String     {The translated error message}
                        ) : boolean          {True if translation was successful}
```

A.5 From PascalInit

```
var
      InPorts           : ptrPortArray;            { Ports inherited from parent }

      TimePort          : port;                    { Time service }
      SesPort           : port;                    { Sesame Server }
      EMPort            : port;                    { Environment manager }

      PMPort            : port;                    { Process Manager }
      NameServerPort    : port;                    { Messgae-Name server }
```

```
          UserTypescript      : Port;                       { This process' typescript }
          UserWindow          : Port;                       { This process' window }
```

A.6 From MsgN from MsgnUser

```
Function CheckIn          (       ServPort        : Port;       { The NameServer Port }
                                  Portsname       : String;
                                  Signature       : Port;       { Authorization Port }
                                  PortsId         : Port;       { Port being checked in }
                           ) : GeneralReturn;

Function CheckOut         (       ServPort        : Port;
                                  PortsName       : String;
                                  Signature       : Port
                          ) : GeneralReturn;

Function LookUp           (       ServPort        : Port;
                                  Portsname       : String;
                          var     portsId         : Port        { The port being looked up }
                          ) : GeneralReturn;
```

A.7 From SesameDefs

```
type
          File_Data           = pointer;
const
          Path_Name_Size      = 255;
```

A.8 From PathName

```
type
          Path_Name           = string [Path_Name_Size];
Function WriteFile        (var PathName           : Path_Name;
                                  Data            : File_Data;
                                  ByteCount       : long
                          ) : GeneralReturn;

Function ReadFile         (var PathName           : Path_Name;
                           var    Data            : File_Data;
                           var    ByteCount       : long
                          ) : GeneralReturn;
```

A.9 From Spawn

Function Exec

```
(var  ChildKPort      : Port
 var  ChildDPort      : Port;
      ProcessName     : string;
      HisCmdLine      : CommandBlock
) : GeneralReturn;
```

Function Split

```
(var  ChildKPort      : Port;
 var  ChildDPort      : Port
) : GeneralReturn;
```

Function Spawn

```
(var  ChildKPort      : Port;
 var  ChildDPort      : Port;
      ProgName        : APath_Name;
      ProcName        : string;
      HisCmdLine      : CommandBlock;
      DebugIt         : boolean;
      ProtectChild    : boolean;
      SapphConn       : ConnectionInheritance;
      pWindow         : Port;
      pTypeScript     : Port;
      EMConn          : ConnectionInheritance;
      pEMPort         : Port;
      PassedPorts     : ptrPortArray;
      NPorts          : long;
      LoaderDebug     : boolean
) : GeneralReturn;
```

Appendix B. A Simple Example

```
Program example;

{-------------------------------------------------------------}
{  Abstract:
{          Forks a child using spawn.
{          The parent passes a port to the child to which the child has send rights.
{          The child registers a second port with the NameServer.
{          The parent looks up this port upon which it has send rights.
{          The Parent and child send each other messages using these two ports.
{
{-------------------------------------------------------------}

imports AccentType from AccentType      { Accent type definitions. }
imports AccCall from AccCall;           { Uses Send and Receive. }
imports AccInt from AccentUser;         { Uses AllocatePort. }
imports MsgN from MsgNUser;             { Uses CheckIn and LookUp. }
imports Spawn from Spawn;               { Uses Spawn. }
imports Pascalinit from pascalinit;     { To obtain system ports. }
imports nameerrors from nameerrors;     { Name Server Error Codes.}
imports commanddefs from commanddefs; { Null_CommandBlock. }

type myMsgType = record
                   Head : Msg;
                   DataDesc : TypeType;
                   Data : integer
                end;

var ChildKPort, ChildDPort       : port;    { Ports representing the Child. }
    ChildtoParPort               : port;    { Ports for communicating }
    PartoChildPort               : port;    { between child and parent. }
    retval                       : GeneralReturn;
    i                            : integer;

    ports                        : ptrPortArray; { Used to pass ports to the child. }
    ports_Count                  : long;
    MyMsg                        : myMsgType;      { The actual message. }

begin
  retval := AllocatePort (KernelPort, ChildtoParPort, 0);
                { Child will be given send rights on this port. }

  new (ports);
  ports↑[0] := ChildtoParPort;      { The port upon which the child will have
                                      send rights and the parent receive rights. }
  ports_count := 1;
```

```
{ Initialise fields in MyMsg. }
with MyMsg.Head
do begin
   simpleMsg := true;
   MsgSize := 2 * Wordsize (myMsgType);
   MsgType := NORMALMSG;
   ID := 42
   end;
with MyMsg.DataDesc
do begin
   Deallocate := false;
   LongForm := false;
   InLine := true;
   TypeName := TYPEINT16;
   TypeSizeInBits := 16;
   NumObjects := 1
   end;

{ Now Spawn the Child process. }
retval := Spawn (              ChildKPort,
                               ChildDPort,
              {ProgName}       '',              { Don't execute a .RUN file.}
              {ProcName}       'MyChild',       { Name of child process. }
              {HisCmdLine}     Null_CommandBlock,
              {DebugIt}        false,           { Don't debug it. }
              {ProtectChild}   false,
              {SapphConn}      Given,           { Use given window. }
              {pWindow}        UserWindow,
              {pTypeScript}    UserTypeScript,
              {EMConn}         Newone,
              {pEMPort}        NULLPORT,
              {PassedPorts}    ports,           { The array of ports. }
              {NPorts}         Ports_count,
                 {LoaderDebug}   False);

if retval = IsParent
then
     begin
     { Initialise the nameServer user end. }
     initmsgn (nullport);
     { Wait for message from Child. }
     writeln ('Parent: Waiting for message from child.');
     MyMsg.Head.LocalPort := ChildtoParPort;
     retval := Receive (MyMsg.Head, 0, LOCALPT, RECEIVEIT);
     if retval = success
        then writeln ('Parent: Received from Child ', MyMsg.Data);

     { Now look up the port that the child has registered with the NameServer. }
     retval := NameNotCheckedIn;
     while retval = NameNotCheckedIn
           do retval := LookUp (NameServerPort,
                                          'CHILDPORT', PartoChildPort);
        { Now send data back to the child. }
        MyMsg.data := MyMsg.data + 1;
        writeln ('Parent: Sending to child ', mymsg.data);
        MyMsg.Head.RemotePort := PartoChildPort;
        retval := Send (MyMsg.Head, 0, WAIT)
        end
   else
        begin { Child }
        { Initialise the nameServer user end. }
        initmsgn (nullport);
        writeln ('Child: started.');
        retval := AllocatePort (KernelPort, PartoChildPort, 0);
        writeln ('Child: Allocated a port OK.');
        retval := CheckIn (NameServerPort, 'CHILDPORT',
                                 nullport, PartoChildPort);

        write ('What is the input data: ');
        readln (i);
        { Send the data to the Parent. }
        MyMsg.data := i;
        writeln ('Child: Sending to Parent ', MyMsg.data);
        MyMsg.Head.RemotePort := InPorts↑[0]; { The ChildtoParPort }
        retval := send (MyMsg.Head, 0, WAIT);

        { Now wait to receive response from parent. }
        MyMsg.Head.LocalPort := PartoChildPort;
        retval := Receive (MyMsg.head, 0, LOCALPT, RECEIVEIT);
        if retval = success
           then writeln ('Child: Received from Parent ', MyMsg.data)
       end

 { P.S. This program actually works.
   It is available on the SPICE VAX in /usr/rds/src/example.pas. }
 end. { Example }
```

Appendix C. Building a System

This appendix describes how to build a new Accent system. It probably should not be in this part of the manual, but I thought that it was better to write it down somewhere rather than nowhere. (Thanks to Jeff Eppinger for an updated version of this appendix.) It is only relevant to you if you are making changes to the Accent Kernel. Changes to processes such as the process manager or shell can be made independently of the kernel.

To begin, obtain all the necessary SEG files from the SPICE VAX. Update from the following logical names with the switch *-test*. You will need:

- ***accentseg_a*** in a directory such as */sys/user/new/accent.*
- ***bootupseg_a*** in a directory such as */sys/user/new/bootup.*
- ***devicesseg_a*** in a directory such as */sys/user/new/devices.*
- ***libpascalseg_a*** in a directory such as */sys/user/new/libpascal.*
- ***micro<size>seg_a*** in a directory such as */sys/user/new/micro.*
  Substitute *4k*, *16k*, or *perq2* for *<size>*.

Also get the file *[cfs]/usr/spice/dev/accent/src/#/accent.config* (where *#* is the highest numbered directory) and store it on your Perq as */sys/user/new/accent/new.config.*

1. Empty your search path except for *<boot>*.
2. Create the operating system run file, call it *new.run*:
   ```
   path /sys/user/new/accent/
   link -nodef-noinit accent ~new
   ```
3. Create the system initialization file, *system.run*:
   ```
   path /sys/user/new/bootup/
   setsearch ../libpascal, ../devices
   link -nodef-noinit system
   ```
4. Make the boot files, call them *new.**:
   ```
   setsearch -pop=2, current:, ../accent, ../micro
   path /sys/spice/
   makevmboot new x system -<perqtype>
   ```
   where *<perqtype>* is one of *oldperq1*, *oldperq1a* or *perq2*.
5. Finally install your new accent system:
   ```
   bindboot -system=new.boot -interpreter=new.<size>.mboot -bootcharacter=x
   ```
   where *<size>* is *4k*, *16k* or *perq2* depending on your machine.

If you wish to modify the accent kernel or system initialization process, get the sources corresponding to the logical names listed above.